mockingboard

mockingboard

mockingboard

text to speech algorithm

sweet micro systems
150 chestnut street
providence, rhode island
02903

mockingboard

mockingboard

mockingboard™

Dem Verein zum Erhalt klassischer Computer zur Verfügung gestellt, Jun. '23 SectorheaD

mockingboard™

text to speech algorithm

By Ed Servello, Jr.
Documentation by Brad Shipp

sweet micro systems
150 chestnut street, providence, rhode island 02903

© 1982 SWEET MICRO SYSTEMS
150 Chestnut Street
Providence, RI 02903 (401) 273-5333

All rights reserved. No part of this manual may be reproduced without prior written permission of Sweet Micro Systems.

Apple II® and Apple II Plus® are registered trademarks of Apple Computer, Inc.
Votrax® is a registered trademark of Votrax

The information contained in this manual is correct as far as we can determine; however, Sweet Micro Systems reserves the right to make improvements in the product and/or manual at any time without notice.

Package Design by Teresa Level, Providence, RI.

INTRODUCTION

The Text To Speech Algorithm program is the latest enhancement for the MOCKINGBOARD SOUND/SPEECH I and the SPEECH I peripheral boards by Sweet Micro Systems. The program was originally conceived and designed as a utility for outputing synthesized speech in the most convenient and logical manner - input the word as it is spelled, translate the word into phonemes and then speak it correctly. Throughout the development of this program, many enhancements were added. Sweet Micro Systems is now proud to present an educational and entertaining utility program guaranteed to challenge your command of the English language while at the same time, enjoyable to use.

The MOCKINGBOARD TEXT TO SPEECH ALGORITHM compares each letter of the word or phrase typed to a table of relational rules to select the correct phonemes. The table is expandable and the rules can be edited for your personal application or preference. The English language is grammatically one of the most complicated human languages and the countless exceptions make it impossible to create a table which is completely effective. Therefore, the ability to create or alter the tables enables you to produce accurate text to speech conversion for specific applications as well as conserve memory and disk space. Unnecessary rules can be eliminated.

Before reading further or trying any of the examples, please make a backup copy of the Text to Speech Algorithm diskette. Also, the MOCKINGBOARD must be installed in slot 4. Please refer to the section entitled INSTALLATION INSTRUCTIONS in the MOCKINGBOARD User's Manual.

After you have made a back-up copy of the diskette boot the diskette again. At this time you are faced with a "?" on the screen with the cursor flashing. You are now ready to begin your initial experimentation with the Text to Speech program. At this time type in any word or sentence and press return. The computer will then speak what you have typed. If it does not speak at all please make sure you have your Mockingboard Speech I or Sound/Speech I properly

installed in the computer. Insure that: (1) cables are connected to speakers, (2) cables are connected to Mockingboard and (3) the two volume controls, located at the top, right rear of Mockingboard, are turned up. The third knob, located just in front of the volume controls, adjusts the pitch of synthesized speech. Experiment with it. First, turn it away from you. The speech is slow and deep. Next, turn it toward you. The speech quickens. Set the knob at the point most appealing to your ear.

Should you wish to have the computer spell rather than speak simply type in the word "spell" and all words you enter will be spelled rather than spoken. To re-enter the speech mode simply type in "speak" and all words typed will again be spoken. In the event you wish to use the words speak or spell as the first word in a sentence hit the space bar before typing, this avoids changing modes. By now you are familiar with the basics of the program so, please read on. Next you will learn about the flexibility built into this program.

A REVIEW OF THE SPEECH SYNTHESIZER

The Mockingboard Sound/Speech I and Speech I use the Votrax phoneme synthesizer to create speech. Each word in the English language can be divided into different sound units called phonemes. There are 64 of these phonemes in our language, including pauses. These are sent to the Votrax chip and are spoken by the Mockingboard. There is a complete list of the phonemes generated by the Votrax chip on page C-2 of the Mockingboard User's Manual. References will be made to this phoneme chart throughout this manual.

The phoneme system allows any word in English, and most foreign words, to be spoken. To create speech, the user must determine the phonemes necessary for each word and then enter them into a table. This, however, has limited applications. For instance, a talking educational program would have to have on file all the words necessary for its purpose and the author of the program would have had to hand compile them.

ALGORITHM

Algorithm is the method used to solve a problem. There are, generally, many ways to approach the problem of converting text into synthesized speech. The Mockingboard Text To Speech Algorithm utilizes a table of rules to identify the correct phoneme to output. The accuracy of each word spoken depends entirely on the comprehensiveness of the rules table. The advantages of this system are that it takes almost no work to create speech from text. This text to speech system can pronounce any word, and the entire rules table (in its current state) takes less than 8,000 bytes.

Some systems are designed to store words in a dictionary to be looked up and spoken if a match occurs. This has one obvious drawback. If the word does not exist in the dictionary, nothing will be spoken. Also, the task of creating and storing an adequate dictionary from the roughly 600,000 words in the English language would consume tremendous time and memory. The time required to look up a word would be prohibitive as well. Most dictionary synthesizers have a vocabulary of under 1,000 words. While sufficient for certain applications, this method is inadequate for a general-purpose speech synthesis system.

The Mockingboard Text To Speech Algorithm works in much the same way a human does when he/she learns how to pronounce a word. There are certain combinations of letters that represent different sounds. For instance, the words AT and ATE. Both contain the letters /A/ and /T/, but one /A/ is the long /A/ and the other short. The algorithm looks for letters like /E/ that modify the pronunciation of a word, and modifies the phonemes accordingly.

Some pronunciation cannot be picked up with a set of rules. Some words simply do not follow normal pronunciation at all. For instance, the word ONE. It should be spelled WON or WUN. Indeed, following the standard rules for pronunciation, ONE should be pronounced like OWN.

Many combinations of characters, like ONE, exist that require special or unique rules for correct pronunciations. The words AVENUE and AVIARY are very similar, but the /A/ sound in the two is quite different. Rules must be very

carefully worked out to account for these very subtle differences. Occassionally, there will be words that are not pronounced correctly.

The Mockingboard Text To Speech Algorithm is in RAM rather than in ROM for much greater flexibility. Algorithms burned into ROM cannot be modified. Although utilizing RAM requires that the rule table be loaded into memory, you can make changes and additions to the pronunciation rules specific to your application. Also, you may not agree with some of the rules in the current table and wish to change them. In ROM, this would not be possible. Jargons or vocabularies specific to a profession or interest exemplify the usefulness of this method. Most people never use the word AVIARY, but an avid bird watcher most certainly would. The bird enthusiast can add rules to accommodate this word, while others can use the space to accommodate other words.

HOW THE ALGORITHM WORKS

The Text to Speech Algorithm is a series of machine language routines that locate the text to be converted; translate the group of characters, sometimes full words, into phonemes; and output them to the Votrax chip on the Mockingboard.

The word or phrase typed is automatically assigned to the variable MB$ (for MockingBoard) by the input routine on the supplied disk. If you create your own input routine in your program, you must also use the MB$ variable as this is the only variable recognized by the text to speech conversion routine. Once the MB$ variable is located in the Applesoft variable table, the text is loaded. For those who have lower case adapters, the text will be converted to upper case.

Before any phoneme generation takes place, the text associated with MB$ is "categorized." That is, each character in the string is broken down into one of the classification types described below and marked accordingly. The markers are stored in a buffer and denotes the classification type of the corresponding character in MB$.

The classification types and symbols are as follows:

!	Non-Alphabetic Character	!@#$%^&*()_+12345678 90<>?,./:;"'[]{}
"	Any Vowel	AEIOU
#	Front Vowel	EIY
$	Zero or More Consonant	BCDFGHJKLMNPQRSTVWXZ
&	One Consonant	BCDFGHJKLMNPQRSTVWXZ
%	Voiced Consonant*	BDGJLMNRVWZ

*A voiced consonant is one that, when pronounced, requires vibration of the vocal cords.

The phrase:	SOUND IS SWEET
would be marked as:	$""%%!#$!$%##$

The next routine is the actual conversion of text to phoneme code. The text in MB$ is read in, one character at a time. It should be noted that the program will read each word from right to left; that is, it will read the last letter or character of the word first and work towards the beginning of the word. The rule table for the character being read is then located. Within the table are a list of sequentially numbered rules. The characters are compared to the rules in successive order. Each rule attempts to isolate a specific condition for a phoneme. The rule may accomplish this by specifying a particular character or characters to the right or left of the character being read. If the condition exits, the phoneme code associated with this rule is stored in a buffer until a driver routine is called to output the phonemes for the word. These rules may also specify the classification marker assigned to the character. This provides for a more general application such as any consonant to the right of the character being read. A more detailed explanation of the rules follows this section.

The following is a sample Basic program for outputting speech through the text to speech program. As you can see, there is very little programming required to produce speech. You may add speech capabilities to your programs by including the text to speech program. The programs are completely portable and we encourage you to explore and create new applications.

```
 10 D$=CHR$(4): REM ALWAYS A GOOD IDEA.
 20 PRINT D$;"BLOAD T-S2.OBJ"
 30 PRINT D$;"BLOAD RULES2.OBJ"
 40 PRINT D$;"BLOAD RNDX2.OBJ"
 50 PRINT D$;"BLOAD PHIDRV2.OBJ"
 60 POKE 1022,161:POKE 1023,67
 70 INPUT "ENTER SOME TEXT:";MB$
 80 CALL 16800: REM SET UP PHONEME TABLE
 90 CALL 17280: REM SEND TABLE TO CARD
100 GOTO 70
```

Lines 10-50 loads the object codes for the four parts of the Text to Speech Algorithm. The T-S2 is the main program where the text is converted to speech. The RULES2 is the actual rule table and RNDX2 is the index to the rule table. The RNDX2 contains the addresses for each character subtable and directs the program to the correct character subtable. Finally the PHIDRV2 is the phoneme interrupt driver which outputs the phonemes to create speech utilizing the APPLE interrupt structure. Line 60 resets the APPLE interrupt vector and directs it to the interrupt routine of the PHIDRV2. Line 70 merely inputs the variable MB$. Line 80 calls the first part of the algorithm (described above), which simply looks in the Applesoft variable lookup table, finds MB$, gets the string, and translates it into a string of phonemes stored in a buffer. Line 90 calls the second half of the algorithm, which sends the phonemes in the buffer to the card to output speech.

Notice that the program loops back to line 70 BEFORE the text has finished being spoken. If you send two strings to the Text to Speech Algorithm too closely together, the second will halt the first in the middle. To counter this and to make programming with the Text to Speech Algorithm easier, there is a flag that the Text to Speech Algorithm sets when it is through speaking. This flag can be polled to test for completion before a program continues. Location 25 ($19) is used as a "busy" flag. If it contains a value of 255, then the SC-01 is still speaking a phoneme. If the address contains a 0, then the SC-01 is finished speaking the phoneme.

RULE TABLE

Just how is the rule table set up? How does the Text To Speech algorithm interpret those rules? These are questions which we will now address.

First, let's return to a previous example: the word AT and the word ATE. Recall that the ending /E/ caused the long /A/ sound in ATE rather than the short /A/ sound as in AT. But why does the words ATE and HATEFUL have the same long /A/ sound. The /E/ is not at the end of the word. Further examination reveals that words like RAKE and TALE also have that long /A/ sound. In fact, any time that /A/ is followed by a consonant and then /E/, the long /A/ is pronounced.

Will all vowels produce the long /A/ sound? If the vowel /O/ follows, will the sound be the same? The word ATOM does not produce the correct /A/ sound. However, the word LABOR does. You may explore this further or develop a separate rule to deal with the vowel /O/. Since all vowels do not produce the desired results, we can only make a general assumption regarding the vowel /E/.

Based upon the assumptions made, a rule is created to test for the following conditions: If the character read is an /A/ then look one space to the right for a consonant. If it is a consonant, then look one more space to the right and check to see if the vowel /E/ is present. If the word fits the conditions, then pronounce the long /A/ sound. The phoneme code for a long /A/ is 05 or 06, depending on personal preference. Refer to the phoneme table in the user's manual and note that the only difference between these two phonemes is the duration that the sound is spoken.

Another assumption may also be made at this time. If the character is an /A/ and the next character is a consonant, but the character after that is NOT a vowel (as in AT or CHATTER) then pronounce phoneme 2E or 2F, again depending on personal preference.

Obviously this is a sweeping assumption, but you may insert more definitive rules between these two rules as you find relationships which appear to produce the same sounds.

There are, of course, words which march to the beat of its own drum - or in this case, the sound of its own rules. That is, words which cannot be converted into speech with conventional rules. An excellent example of this and one which troubled us greatly, is the /CH/ sound. These two letters can assume many different sounds and apparently follow no particular rule (at least none was discovered). Words with these letters can be pronounced hard, similar to a /K/ sound, as in CHARACTER; or soft, similar to a /SH/ sound, as in CHARADE; not to mention its own sound, as in CHARCOAL. All of these examples begin with CHAR, yet they are pronounced three different ways without a clue as to why. The only way we have found to deal with such exceptions, is to create a rule to look specifically for the word and assign the appropriate phoneme code.

To continue our discussion on developing rules, the vowel /E/ produces some interesting and unique conditions.

> If the next character is also an /E/, then send phoneme 2C, and skip one character in the text. Why skip a character? Because the letter group EE corresponds to only one particular sound, not two separate sounds.

Sometimes the /E/ is not pronounced but used instead to modify the pronunciation of another syllable. In the word ATE, for instance, the /E/ is not pronounced, but is used merely to indicate the correct pronunciation of the /A/.

> If an /E/ is encountered and the previous character was a consonant, and the character before that was a vowel, then do not pronounce anything.

The word WHY is another good example of words that follow no easily definable general rule - so it has a rule of its own.

> If the first character is a /W/, the next is an /H/, and the one following that is a /Y/, then skip three characters and pronounce the phonemes 2D, 08 and 22.

You can see that making rules that are effective and efficient is no simple matter. The only way to become proficient at developing rules is to actually do it

yourself. A program on the Text To Speech Algorithm disk called PRE2 (an acronym for Phoneme Rule Editor, version 2) was written specifically to make this task as simple as possible. PRE2 is used to create, test and edit rules. Of course, there are many rules already on the disk, but you may wish to add rules, or even delete some.

PHONEME RULE EDITOR

The Text To Speech Algorithm's input program is automatically loaded at the beginning when the disk is booted. To access the PRE2 program:

1. Type QUIT next to the /?/ to exit the program
2. Type RUN PRE2
3. A menu of commands will appear. Hit the space bar.
4. Enter the character subtable to be edited. (A set of asterisks will blink back and forth to let you know the table is loading correctly.)
5. The components of the subtable will be displayed on the screen. (Don't worry, they are supposed to look sort of strange at first.)

For the purpose of demonstration, the rule table for /A/ will be used. You should see something like this:

```
 1    >!=0131
 2    >IR!=0220
 3    !>RE!=0115
 4    >NGE!=012022
 5    $>L!=023318
        :
        :
        :
12    >&ION!=0120
        :
27    =012F
```

As you may have guessed, the rule table is decoded into regular characters, each of which have a certain meaning. Look at RULE #1: [>!=0131]

[>] "look at characters to the right of /A/"
[!] "look for a non-alphabetic character, like a space, a period, or a comma"
[=] signifies the start of the phoneme table for that rule.
[01] The next two digits tell the program how many characters to skip over in the literal string before trying to interpret the next character. In this example, skip only the /A/.
[31] The next group of numbers are the phoneme codes in HEX. Each phoneme is two digits long. There can be as many as necessary. In this example, phoneme 31 (UH2) is sent to the chip for output.

Translated into English, this rule reads something like "If the next character is non-alphabetic (a space or period), skip one character, and send phoneme 31 to the card to be spoken." The last /A/ in ALPHA would trigger this rule.
RULE #3: [!>RE!=0115]

[!] "look for a non-alphabetic character in front of the character /A/"
[>RE] "look for RE to the right of /A/"
[!] "look for a non-alphabetic character next"

Rule number 3 reads, "If the character before the /A/ is non-alphabetic (ie- /A/ is the first letter in the word), and /A/ is followed by RE and then another non-alphabetic (signifying the end of the word), then skip 01 characters and pronounce phoneme code 15 (AH1)." If you have been paying particularly close attention, you may have noticed that this is a rule specific to the word ARE. Notice that although the entire word is recognized within the /A/ rule table, the rest of the word (RE) is pronounced within other rule tables. We could just as easily have skipped over 3 characters and sent the phoneme codes for the entire word ARE (code 152B for AH1R) to the card.

You may be wondering why ARE must have its own rule. Why not just check for words ending in RE? Then words such as HARE or CARE would be pronounced incorrectly. ARE is similar to ONE - it just doesn't follow any of the rules, so it has to have one if its own.

Take a look at RULE #12: [>&ION!=0120].
A new symbol, the [&], is used to denote ONE consonant. This

rule looks for words ending in /A/-/ION/, where - is a consonant. AUTOMATION and VACATION are words which would match this rule.

Other symbols of importance are:
[$] looks for zero or more consonants or any number of consonant
["] looks for any vowel
[#] looks for the vowels E, I and Y
There is no character for multiple vowels.

The rule [>"=012F] looks for any vowel to the right of /A/ such as AE, AI, AO, AU, etc.

Below is a quick summary of the character matching symbols:

!	Non-alphabetic character
$	Zero or more consonants
&	One consonant
"	Any vowels
#	E, Y, I
>	Search to right of character

Rules are evaluated sequentially until the word matches the conditions set by a rule. Rule #1 will be checked first, then 2, etc. Therefore, it is important to carefully enter the rules in the order you wish the word to be evaluated. For instance, a rule reading [A>R=0102] to check for occurances of AAR would be ignored if preceeded by [">AR=022D1E], to check for an AR preceeded by ANY vowel. Improper location of rules can make a difference.

Creating a Rule

Let's try making a rule of our own, but on paper. As mentioned earlier, the word ONE follows no rules. Therefore, a rule must be made especially for it. First, the location of the rule must be determined. The O rule table would be the most logical place.

Since we are looking for just the word ONE and not the three letters /O/, /N/, /E/ (as in DONE), our rule must look for the beginning and end of the word. The [!] character can be

used to detect both the beginning and end of the word. To the left of the [=], our rule would look like this:

!>NE!

"If the character to the left of the O is a space, period, comma, etc., and to the right are the letters N and E, followed by another non-alphabetic character (signifying the end of the word), then..."

Now we have to choose some phonemes to represent the word we are speaking. Since the rule looks for the entire word, we must choose phonemes for the entire word. The phonemes 2D, 33 and 0D will pronounce the word ONE.

!>NE!=XX2D330D

The XX is the number of characters that we wish the Text to Speech Algorithm to skip over in the literal string. Since we have interpreted three characters in this rule, we want the algorithm to move forward three characters. The XX becomes 03.

!>NE!=032D330D

If you look in the O table, you will find a rule that looks very much like that one.

Editing Commands

The following commands are available for editing and adding rules:

S: Select a Rule Table - Switch from one character's rule table to those of another.

I: Insert a Rule - Allows a rule to be inserted before another within a specific rule table.

D: Delete a Rule

C: Change a Rule - This is the same as deleting and then inserting a new rule.

E: Evaluate Rules - Drops you into the test mode. A [>] prompt will appear. This lets you test the rules that you have just modified.

A: Archive Rules to Disk - BSAVE the rule table. The rule table is saved in a file called RULES2.BAK. This is a precautionary measure to allow the user to reuse the original rule table.

M: Memory Update - Updates the rule table in memory, making permanent any changes that you have entered. Unless you select the Memory update option, any changes that you have made will be lost, so be sure and update memory before evaluating a rule, selecting a new one, or archiving to disk.

?: Display Menu

ESC: Exit Program

Many times, especially when dealing with exceptions to rules, the phoneme for a particular character may be sent along with those of the last character evaluated. An example of this would be the /WH/ words like WHY and WHAT. Since WHAT is hard to distinguish between HAT and CAT, but is pronounced differently, the /A/ sound in WHAT is stored along with the /W/ rules. One of the /W/ rules looks for a /H/ and an /A/, and then sends the phonemes for all of them together. Keep this in mind when editing rules.

There is a maximum size of 256 characters per character subtable. This includes all the special characters used to formulate the rules.

NOTE: It is a good idea to make a backup copy of your original rule table before trying to modify it.

INSTRUCTIONS FOR SAVING EDITED RULE TABLES TO DISK

After editing a rule, executing a Memory Update and evaluating the change, you must save the rule tables to disk. The rule tables are saved to disk by executing an Archive Rules to Disk command. You will be asked if it is to be saved as Rules2.Bak. Rules2.Bak is a backup for the rule tables. This is a precautionary measure to insure you do not overwrite the rule tables in error. If you are satisfied that the changes should be made permanent, delete Rules2.Obj, rename Rules2.Bak to Rules2.Obj. If you did not intend to save the tables to disk, just delete Rules2.Bak.